


基础概念与常识

Java 语言有哪些特点?

1. 简单易学;
2. 面向对象 (封装, 继承, 多态) ;
3. 平台无关性 (Java 虚拟机实现平台无关性) ;
4. 支持多线程 (C++ 语言没有内置的多线程机制, 因此必须调用操作系统的多线程功能来进行多线程程序设计, 而 Java 语言却提供了多线程支持) ;
5. 可靠性;
6. 安全性;
7. 支持网络编程并且很方便 (Java 语言诞生本身就是为简化网络编程设计的, 因此 Java 语言不仅支持网络编程而且很方便) ;
8. 编译与解释并存;

 **修正 (参见: [issue#544](#))** : C++11 开始 (2011 年的时候) ,C++就引入了多线程库, 在 windows、linux、macos 都可以使用`std::thread`和`std::async`来创建线程。参考链接:
<http://www.cplusplus.com/reference/thread/thread/?kw=thread>

 拓展一下:

“Write Once, Run Anywhere (一次编写, 随处运行)”这句宣传口号, 真心经典, 流传了好多年! 以至于, 直到今天, 依然有很多人觉得跨平台是 Java 语言最大的优势。实际上, 跨平台已经不是 Java 最大的卖点了, 各种 JDK 新特性也不是。目前市面上虚拟化技术已经非常成熟, 比如你通过 Docker 就很容易实现跨平台了。在我看来, Java 强大的生态才是!

JVM vs JDK vs JRE

JVM

Java 虚拟机 (JVM) 是运行 Java 字节码的虚拟机。JVM 有针对不同系统的特定实现 (Windows, Linux, macOS), 目的是使用相同的字节码, 它们都会给出相同的结果。字节码和不同系统的 JVM 实现是 Java 语言“一次编译, 随处可以运行”的关键所在。

JVM 并不是只有一种! 只要满足 JVM 规范, 每个公司、组织或者个人都可以开发自己的专属 JVM。 也就是说我们平时接触到的 HotSpot VM 仅仅是是 JVM 规范的一种实现而已。

除了我们平时最常用的 HotSpot VM 外, 还有 J9 VM、Zing VM、JRockit VM 等 JVM。维基百科上就有常见 JVM 的对比: [Comparison of Java virtual machines](#), 感兴趣的可以去看看。并且, 你可以在 [Java SE Specifications](#) 上找到各个版本的 JDK 对应的 JVM 规范。



Java SE > Java SE Specifications

Java Language and Virtual Machine Specifications

Java SE 18

Released March 2022 as [JSR 393](#)

The Java Language Specification, Java SE 18 Edition

- [HTML](#) | [PDF](#)
- Preview feature: [Pattern Matching for switch](#)



The Java Virtual Machine Specification, Java SE 18 Edition

- [HTML](#) | [PDF](#)

Java SE 17

Released September 2021 as [JSR 392](#)

The Java Language Specification, Java SE 17 Edition

- [HTML](#) | [PDF](#)
- Preview feature: [Pattern Matching for switch](#)



The Java Virtual Machine Specification, Java SE 17 Edition

- [HTML](#) | [PDF](#)

JDK 和 JRE

JDK 是 Java Development Kit 缩写，它是功能齐全的 Java SDK。它拥有 JRE 所拥有的一切，还有编译器（javac）和工具（如 javadoc 和 jdb）。它能够创建和编译程序。

JRE 是 Java 运行时环境。它是运行已编译 Java 程序所需的所有内容的集合，包括 Java 虚拟机（JVM），Java 类库，java 命令和其他的一些基础构件。但是，它不能用于创建新程序。

如果你只是为了运行一下 Java 程序的话，那么你只需要安装 JRE 就可以了。如果你需要进行一些 Java 编程方面的工作，那么你就需要安装 JDK 了。但是，这不是绝对的。有时，即使您不打算在计算机上进行任何 Java 开发，仍然需要安装 JDK。例如，如果要使用 JSP 部署 Web 应用程序，那么从技术上讲，您只是在应用程序服务器中运行 Java 程序。那你为什么需要 JDK 呢？因为应用程序服务器会将 JSP 转换为 Java servlet，并且需要使用 JDK 来编译 servlet。

什么是字节码？采用字节码的好处是什么？

在 Java 中，JVM 可以理解的代码就叫做字节码（即扩展名为 `.class` 的文件），它不面向任何特定的处理器，只面向虚拟机。Java 语言通过字节码的方式，在一定程度上解决了传统解释型语言执行效率低的问题，同时又保留了解释型语言可移植的特点。所以，Java 程序运行时相对来说还是高效的（不过，和 C++，Rust，Go 等语言还是有一定差距的），而且，由于字节码并不针对一种特定的机器，因此，Java 程序无须重新编译便可在多种不同操作系统的计算机上运行。

Java 程序从源代码到运行的过程如下图所示：

公众号: JavaGuide
网站: javaguide.cn



我们需要格外注意的是 `.class`→`机器码` 这一步。在这一步 JVM 类加载器首先加载字节码文件，然后通过解释器逐行解释执行，这种方式的执行速度会相对比较慢。而且，有些方法和代码块是经常需要被调用的(也就是所谓的热点代码)，所以后面引进了 JIT (just-in-time compilation) 编译器，而 JIT 属于运行时编译。当 JIT 编译器完成第一次编译后，其会将字节码对应的机器码保存下来，下次可以直接使用。而我们知道，机器码的运行效率肯定是高于 Java 解释器的。这也解释了我们为什么经常会说 **Java 是编译与解释共存的语言**。

HotSpot 采用了惰性评估(Lazy Evaluation)的做法，根据二八定律，消耗大部分系统资源的只有那一小部分的代码（热点代码），而这也就是 JIT 所需要编译的部分。JVM 会根据代码每次被执行的情况收集信息并相应地做出一些优化，因此执行的次数越多，它的速度就越快。JDK 9 引入了一种新的编译模式 AOT(Ahead of Time Compilation)，它是直接将字节码编译成机器码，这样就避免了 JIT 预热等各方面的开销。JDK 支持分层编译和 AOT 协作使用。

为什么不全部使用 AOT 呢？

AOT 可以提前编译节省启动时间，那为什么不全部使用这种编译方式呢？

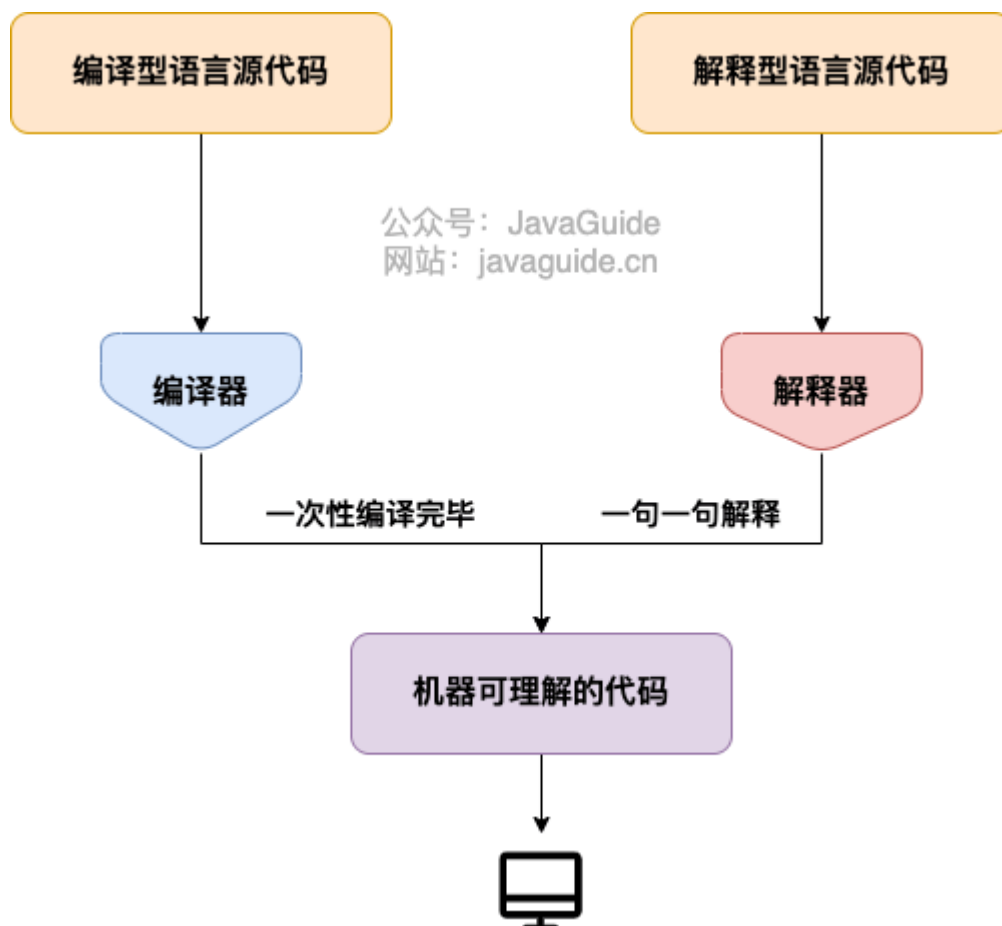
长话短说，这和 Java 语言的动态特性有千丝万缕的联系了。举个例子，CGLIB 动态代理使用的是 ASM 技术，而这种技术大致原理是运行时直接在内存中生成并加载修改后的字节码文件也就是 `.class` 文件，如果全部使用 AOT 提前编译，也就不能使用 ASM 技术了。为了支持类似的动态特性，所以选择使用 JIT 即时编译器。

为什么说 Java 语言“编译与解释并存”？

其实这个问题我们讲字节码的时候已经提到过，因为比较重要，所以我们这里再提一下。

我们可以将高级编程语言按照程序的执行方式分为两种：

- **编译型**：**编译型语言** 会通过**编译器**将源代码一次性翻译成可被该平台执行的机器码。一般情况下，编译语言的执行速度比较快，开发效率比较低。常见的编译性语言有 C、C++、Go、Rust 等等。
- **解释型**：**解释型语言**会通过**解释器**一句一句的将代码解释（interpret）为机器代码后再执行。解释型语言开发效率比较快，执行速度比较慢。常见的解释性语言有 Python、JavaScript、PHP 等等。



根据维基百科介绍:

为了改善编译语言的效率而发展出的**即时编译**技术, 已经缩小了这两种语言间的差距。这种技术混合了编译语言与解释型语言的优点, 它像编译语言一样, 先把程序源代码编译成**字节码**。到执行期时, 再将字节码直译, 之后执行。**Java**与**LLVM**是这种技术的代表产物。

相关阅读: [基本功](#) | [Java 即时编译器原理解析及实践](#)

为什么说 Java 语言“编译与解释并存”?

这是因为 Java 语言既具有编译型语言的特征, 也具有解释型语言的特征。因为 Java 程序要经过先编译, 后解释两个步骤, 由 Java 编写的程序需要先经过编译步骤, 生成字节码 (`.class` 文件), 这种字节码必须由 Java 解释器来解释执行。

Oracle JDK vs OpenJDK

可能在看这个问题之前很多人和我一样并没有接触和使用过 OpenJDK。那么 Oracle JDK 和 OpenJDK 之间是否存在重大差异? 下面我通过收集到的一些资料, 为你解答这个被很多人忽视的问题。

对于 Java 7, 没什么关键的地方。OpenJDK 项目主要基于 Sun 捐赠的 HotSpot 源代码。此外, OpenJDK 被选为 Java 7 的参考实现, 由 Oracle 工程师维护。关于 JVM, JDK, JRE 和 OpenJDK 之间的区别, Oracle 博客帖子在 2012 年有一个更详细的答案:

问: OpenJDK 存储库中的源代码与用于构建 Oracle JDK 的代码之间有什么区别?

答: 非常接近 - 我们的 Oracle JDK 版本构建过程基于 OpenJDK 7 构建, 只添加了几个部分, 例如部署代码, 其中包括 Oracle 的 Java 插件和 Java WebStart 的实现, 以及一些闭源的第三方组件, 如图形光栅化器, 一些开源的第三方组件, 如 Rhino, 以及一些零碎的东西, 如附加文档或第三方字体。展望未来, 我们的目的是开源 Oracle JDK 的所有部分, 除了我们考虑商业功能的部分。

总结：（提示：下面括号内的内容是基于原文补充说明的，因为原文太过于晦涩难懂，用人话重新解释了下，如果你看得懂里面的术语，可以忽略括号解释的内容）

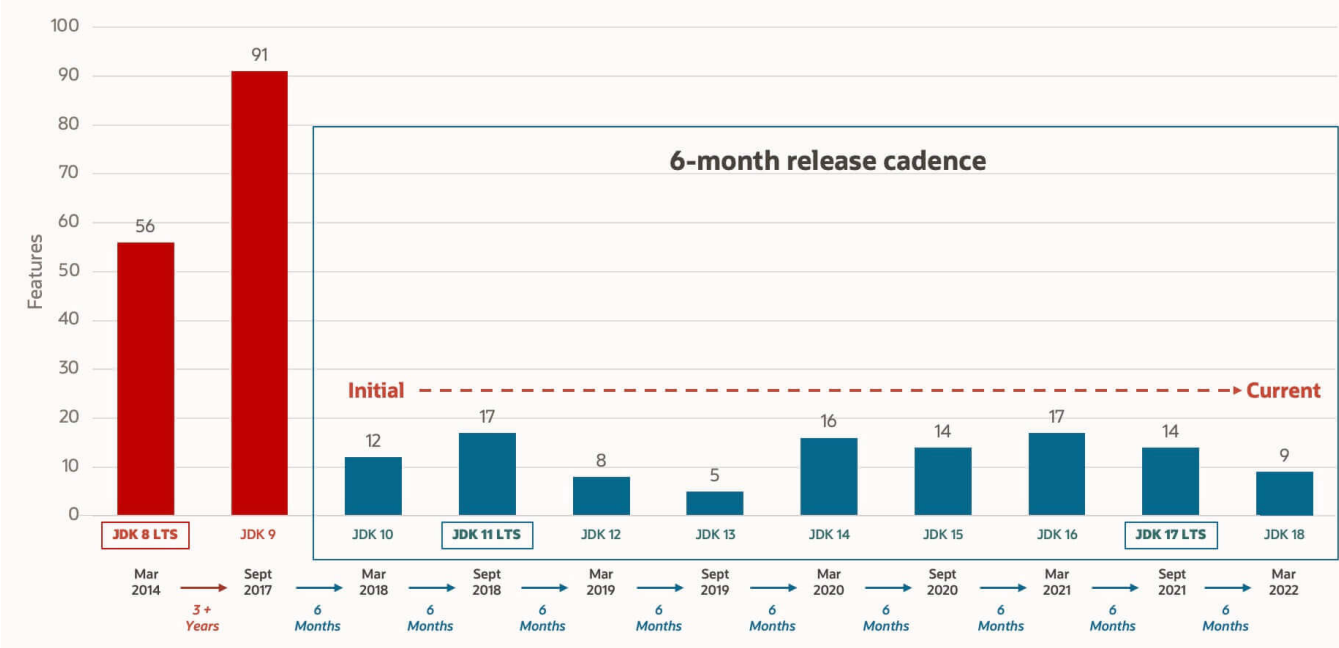
1. Oracle JDK 大概每 6 个月发一次主要版本（从 2014 年 3 月 JDK 8 LTS 发布到 2017 年 9 月 JDK 9 发布经历了长达 3 年多的时间，所以并不总是 6 个月），而 OpenJDK 版本大概每三个月发布一次。但这不是固定的，我觉得了解这个没啥用处。详情参见：<https://blogs.oracle.com/java-platform-group/update-and-faq-on-the-java-se-release-cadence>。
2. OpenJDK 是一个参考模型并且是完全开源的，而 Oracle JDK 是 OpenJDK 的一个实现，并不是完全开源的；（个人观点：众所周知，JDK 原来是 SUN 公司开发的，后来 SUN 公司又卖给了 Oracle 公司，Oracle 公司以 Oracle 数据库而著名，而 Oracle 数据库又是闭源的，这个时候 Oracle 公司就不想完全开源了，但是原来的 SUN 公司又把 JDK 给开源了，如果这个时候 Oracle 收购回来之后就把他给闭源，必然会引起很多 Java 开发者的不满，导致大家对 Java 失去信心，那 Oracle 公司收购回来不就把 Java 烂在手里了吗！然后，Oracle 公司就想了个骚操作，这样吧，我把一部分核心代码开源出来给你们玩，并且我要和你们自己搞的 JDK 区分下，你们叫 OpenJDK，我叫 Oracle JDK，我发布我的，你们继续玩你们的，要是你们搞出来什么好玩的东西，我后续发布 Oracle JDK 也会拿来用一下，一举两得！）OpenJDK 开源项目：<https://github.com/openjdk/jdk>
3. Oracle JDK 比 OpenJDK 更稳定（肯定啦，Oracle JDK 由 Oracle 内部团队进行单独研发的，而且发布时间比 OpenJDK 更长，质量更有保障）。OpenJDK 和 Oracle JDK 的代码几乎相同（OpenJDK 的代码是从 Oracle JDK 代码派生出来的，可以理解为在 Oracle JDK 分支上拉了一条新的分支叫 OpenJDK，所以大部分代码相同），但 Oracle JDK 有更多的类和一些错误修复。因此，如果您想开发企业/商业软件，我建议您选择 Oracle JDK，因为它经过了彻底的测试和稳定。某些情况下，有些人提到在使用 OpenJDK 可能会遇到了许多应用程序崩溃的问题，但是，只需切换到 Oracle JDK 就可以解决问题；
4. 在响应性和 JVM 性能方面，Oracle JDK 与 OpenJDK 相比提供了更好的性能；
5. Oracle JDK 不会为即将发布的版本提供长期支持（如果是 LTS 长期支持版本的话也会，比如 JDK 8，但并不是每个版本都是 LTS 版本），用户每次都必须通过更新到最新版本获得支持来获取最新版本；
6. Oracle JDK 使用 BCL/OTN 协议获得许可，而 OpenJDK 根据 GPL v2 许可获得许可。

既然 Oracle JDK 这么好，那为什么还要有 OpenJDK？

答：

1. OpenJDK 是开源的，开源意味着你可以对它根据你自己的需要进行修改、优化，比如 Alibaba 基于 OpenJDK 开发了 Dragonwell8：<https://github.com/alibaba/dragonwell8>
2. OpenJDK 是商业免费的（这也是为什么通过 yum 包管理器上默认安装的 JDK 是 OpenJDK 而不是 Oracle JDK）。虽然 Oracle JDK 也是商业免费（比如 JDK 8），但并不是所有版本都是免费的。
3. OpenJDK 更新频率更快。Oracle JDK 一般是每 6 个月发布一个新版本，而 OpenJDK 一般是每 3 个月发布一个新版本。（现在你知道为啥 Oracle JDK 更稳定了吧，先在 OpenJDK 试试水，把大部分问题都解决掉了才在 Oracle JDK 上发布）

基于以上这些原因，OpenJDK 还是有存在的必要的！



拓展一下：

- BCL 协议（Oracle Binary Code License Agreement）： 可以使用 JDK（支持商用），但是不能进行修改。
- OTN 协议（Oracle Technology Network License Agreement）： 11 及之后新发布的 JDK 用的都是这个协议，可以自己私下用，但是商用需要付费。

Oracle JDK 各个版本所用的协议		
Oracle JDK 版本	BCL协议	OTN协议
6	最后一个公共更新6u45之前	
7	最后一个公共更新7u80之前	
8	8u201/8u202之前	8u211/8u212之后
9	✓	
10	✓	
11		✓
12		✓

相关阅读 📖： [《Differences Between Oracle JDK and OpenJDK》](#)

Java 和 C++ 的区别？

我知道很多人没学过 C++，但是面试官就是没事喜欢拿咱们 Java 和 C++ 比呀！没办法！！！就算没学过 C++，也要记下来。

虽然，Java 和 C++ 都是面向对象的语言，都支持封装、继承和多态，但是，它们还是有挺多不相同的地方：

- Java 不提供指针来直接访问内存，程序内存更加安全
- Java 的类是单继承的，C++ 支持多重继承；虽然 Java 的类不可以多继承，但是接口可以多继承。

- Java 有自动内存管理垃圾回收机制(GC)，不需要程序员手动释放无用内存。
- C++同时支持方法重载和操作符重载，但是 Java 只支持方法重载（操作符重载增加了复杂性，这与 Java 最初的设计思想不符）。
-

基本语法

注释有哪几种形式？

Java 中的注释有三种：

1. **单行注释**：通常用于解释方法内某单行代码的作用。
2. **多行注释**：通常用于解释一段代码的作用。
3. **文档注释**：通常用于生成 Java 开发文档。

用的比较多的还是单行注释和文档注释，多行注释在实际开发中使用的相对较少。

```
/**
 * Return whether this cache manager stores a copy of each entry or
 * a reference for all its caches. If store by value is enabled, any
 * cache entry must be serializable.
 * @since 4.3
 */
public boolean isStoreByValue() {
    return this.storeByValue;
}

@Override
public void setBeanClassLoader(ClassLoader classLoader) {
    this.serialization = new SerializationDelegate(classLoader);
    // Need to recreate all Cache instances with new ClassLoader in store-by-value mode...
    if (isStoreByValue()) {
        recreateCaches();
    }
}
```

文档注释

单行注释

在我们编写代码的时候，如果代码量比较少，我们自己或者团队其他成员还可以很轻易地看懂代码，但是当项目结构一旦复杂起来，我们就需要用到注释了。注释并不会执行(编译器在编译代码之前会把代码中的所有注释抹掉,字节码中不保留注释)，是我们程序员写给自己看的，注释是你的代码说明书，能够帮助看代码的人快速地理清代码之间的逻辑关系。因此，在写程序的时候随手加上注释是一个非常好的习惯。

《Clean Code》这本书明确指出：

代码的注释不是越详细越好。实际上好的代码本身就是注释，我们要尽量规范和美化自己的代码来减少不必要的注释。

若编程语言足够有表达力，就不需要注释，尽量通过代码来阐述。

举个例子：

去掉下面复杂的注释，只需要创建一个与注释所言同一事物的函数即可

```
// check to see if the employee is eligible for full benefits
if ((employee.flags & HOURLY_FLAG) && (employee.age > 65))
```

应替换为

```
if (employee.isEligibleForFullBenefits())
```

标识符和关键字的区别是什么？

在我们编写程序的时候，需要大量地为程序、类、变量、方法等取名字，于是就有了 **标识符**。简单来说，**标识符就是一个名字**。

有一些标识符，Java 语言已经赋予了其特殊的含义，只能用于特定的地方，这些特殊的标识符就是 **关键字**。简单来说，**关键字是被赋予特殊含义的标识符**。比如，在我们的日常生活中，如果我们想要开一家店，则要给这个店起一个名字，起的这个“名字”就叫标识符。但是我们店的名字不能叫“警察局”，因为“警察局”这个名字已经被赋予了特殊的含义，而“警察局”就是我们日常生活中的关键字。

Java 语言关键字有哪些？

分类	关键字						
访问控制	private	protected	public				
类，方法和变量修饰符	abstract	class	extends	final	implements	interface	native
	new	static	strictfp	synchronized	transient	volatile	enum
程序控制	break	continue	return	do	while	if	else
	for	instanceof	switch	case	default	assert	
错误处理	try	catch	throw	throws	finally		
包相关	import	package					
基本类型	boolean	byte	char	double	float	int	long
	short						
变量引用	super	this	void				
保留字	goto	const					

Tips：所有的关键字都是小写的，在 IDE 中会以特殊颜色显示。

`default` 这个关键字很特殊，既属于程序控制，也属于类，方法和变量修饰符，还属于访问控制。

- 在程序控制中，当在 `switch` 中匹配不到任何情况时，可以使用 `default` 来编写默认匹配的情况。
- 在类，方法和变量修饰符中，从 JDK8 开始引入了默认方法，可以使用 `default` 关键字来定义一个方法的默认实现。
- 在访问控制中，如果一个方法前没有任何修饰符，则默认会有一个修饰符 `default`，但是这个修饰符加上了就会报错。

⚠ 注意：虽然 `true`, `false`, 和 `null` 看起来像关键字但实际上他们是字面值，同时你也不可以作为标识符来使用。

官方文档：https://docs.oracle.com/javase/tutorial/java/nutsandbolts/_keywords.html

自增自减运算符

在写代码的过程中，常见的一种情况是需要某个整数类型变量增加 1 或减少 1，Java 提供了一种特殊的运算符，用于这种表达式，叫做自增运算符 (`++`) 和自减运算符 (`--`)。

`++` 和 `--` 运算符可以放在变量之前，也可以放在变量之后，当运算符放在变量之前时(前缀)，先自增/减，再赋值；当运算符放在变量之后时(后缀)，先赋值，再自增/减。例如，当 `b = ++a` 时，先自增（自己增加 1），再赋值（赋值给 b）；当 `b = a++` 时，先赋值（赋值给 b），再自增（自己增加 1）。也就是，`++a` 输出的是 `a+1` 的值，`a++` 输出的是 `a` 值。用一句口诀就是：“符号在前就先加/减，符号在后就后加/减”。

移位运算符

移位运算符是最基本的运算符之一，几乎每种编程语言都包含这一运算符。移位操作中，被操作的数据被视为二进制数，移位就是将其向左或向右移动若干位的运算。

移位运算符在各种框架以及 JDK 自身的源码中使用还是挺广泛的，`HashMap` (JDK1.8) 中的 `hash` 方法的源码就用到了移位运算符：

```
static final int hash(Object key) {
    int h;
    // key.hashCode(): 返回散列值也就是hashcode
    // ^ : 按位异或
    // >>>: 无符号右移，忽略符号位，空位都以0补齐
    return (key == null) ? 0 : (h = key.hashCode()) ^ (h >>> 16);
}
```

在 Java 代码里使用 `<<`、`>>` 和 `>>>` 转换成的指令码运行起来会更高效率些。

掌握最基本的移位运算符知识还是很有必要的，这不光可以帮助我们在代码中使用，还可以帮助我们理解源码中涉及到移位运算符的代码。

Java 中有三种移位运算符：

- `<<`: 左移运算符，向左移若干位，高位丢弃，低位补零。`x << 1` 相当于 `x` 乘以 2 (不溢出的情况下)。
- `>>`: 带符号右移，向右移若干位，高位补符号位，低位丢弃。正数高位补 0，负数高位补 1。`x >> 1` 相当于 `x` 除以 2。
- `>>>`: 无符号右移，忽略符号位，空位都以 0 补齐。

由于 `double`, `float` 在二进制中的表现比较特殊，因此不能来进行移位操作。

移位操作符实际上支持的类型只有 `int` 和 `long`，编译器在对 `short`、`byte`、`char` 类型进行移位前，都会将其转换为 `int` 类型再操作。

如果移位的位数超过数值所占有的位数会怎样？

当 `int` 类型左移/右移位数大于等于 32 位操作时，会先求余 (%) 后再进行左移/右移操作。也就是说左移/右移 32 位相当于不进行移位操作 (`32%32=0`)，左移/右移 42 位相当于左移/右移 10 位 (`42%32=10`)。当 `long` 类型进行左移/右移操作时，由于 `long` 对应的二进制是 64 位，因此求余操作的基数也变成了 64。

也就是说：`x<<42` 等同于 `x<<10`，`x>>42` 等同于 `x>>10`，`x >>>42` 等同于 `x >>> 10`。

左移运算符代码示例：

```
int i = -1;
System.out.println("初始数据:  " + i);
System.out.println("初始数据对应的二进制字符串:  " +
Integer.toBinaryString(i));
i <<= 10;
System.out.println("左移 10 位后的数据 " + i);
System.out.println("左移 10 位后的数据对应的二进制字符 " +
Integer.toBinaryString(i));
```

输出：

```
初始数据:  -1
初始数据对应的二进制字符串:  11111111111111111111111111111111
左移 10 位后的数据 -1024
左移 10 位后的数据对应的二进制字符 1111111111111111111110000000000
```

由于左移位数大于等于 32 位操作时，会先求余 (%) 后再进行左移操作，所以下面的代码左移 42 位相当于左移 10 位 ($42\%32=10$)，输出结果和前面的代码一样。

```
int i = -1;
System.out.println("初始数据:  " + i);
System.out.println("初始数据对应的二进制字符串:  " +
Integer.toBinaryString(i));
i <<= 42;
System.out.println("左移 10 位后的数据 " + i);
System.out.println("左移 10 位后的数据对应的二进制字符 " +
Integer.toBinaryString(i));
```

右移运算符使用类似，篇幅问题，这里就不做演示了。

continue、break 和 return 的区别是什么？

在循环结构中，当循环条件不满足或者循环次数达到要求时，循环会正常结束。但是，有时候可能需要在循环的过程中，当发生了某种条件之后，提前终止循环，这就需要用到下面几个关键词：

1. `continue`：指跳出当前的这一次循环，继续下一次循环。
2. `break`：指跳出整个循环体，继续执行循环下面的语句。

`return` 用于跳出所在方法，结束该方法的运行。`return` 一般有两种用法：

1. `return;`：直接使用 `return` 结束方法执行，用于没有返回值函数的方法
2. `return value;`：`return` 一个特定值，用于有返回值函数的方法

思考一下：下列语句的运行结果是什么？

```
public static void main(String[] args) {
    boolean flag = false;
    for (int i = 0; i <= 3; i++) {
        if (i == 0) {
            System.out.println("0");
        } else if (i == 1) {
            System.out.println("1");
            continue;
        } else if (i == 2) {
            System.out.println("2");
            flag = true;
        } else if (i == 3) {
            System.out.println("3");
            break;
        } else if (i == 4) {
            System.out.println("4");
        }
        System.out.println("xixi");
    }
    if (flag) {
        System.out.println("haha");
        return;
    }
    System.out.println("heihei");
}
```

运行结果：

```
0
xixi
1
2
xixi
3
haha
```

变量

成员变量与局部变量的区别？

- **语法形式**：从语法形式上看，成员变量是属于类的，而局部变量是在代码块或方法中定义的变量或是方法的参数；成员变量可以被 `public,private,static` 等修饰符所修饰，而局部变量不能被访问控制修饰符及 `static` 所修饰；但是，成员变量和局部变量都能被 `final` 所修饰。
- **存储方式**：从变量在内存中的存储方式来看,如果成员变量是使用 `static` 修饰的，那么这个成员变量是属于类的，如果没有使用 `static` 修饰，这个成员变量是属于实例的。而对象存在于堆内存，局部变量则存在于栈内存。
- **生存时间**：从变量在内存中的生存时间上看，成员变量是对象的一部分，它随着对象的创建而存在，而局部变量随着方法的调用而自动生成，随着方法的调用结束而消亡。

- **默认值**：从变量是否有默认值来看，成员变量如果没有被赋初始值，则会自动以类型的默认值而赋值（一种情况例外：被 `final` 修饰的成员变量也必须显式地赋值），而局部变量则不会自动赋值。

静态变量有什么作用？

静态变量可以被类的所有实例共享。无论一个类创建了多少个对象，它们都共享同一份静态变量。

通常情况下，静态变量会被 `final` 关键字修饰成为常量。

字符型常量和字符串常量的区别？

1. **形式**：字符常量是单引号引起的一个字符，字符串常量是双引号引起的 0 个或若干个字符。
2. **含义**：字符常量相当于一个整型值(ASCII 值),可以参加表达式运算; 字符串常量代表一个地址值(该字符串在内存中存放位置)。
3. **占内存大小**：字符常量只占 2 个字节; 字符串常量占若干个字节。

(注意： `char` 在 Java 中占两个字节)

方法

什么是方法的返回值?方法有哪几种类型?

方法的返回值 是指我们获取到的某个方法体中的代码执行后产生的结果！（前提是该方法可能产生结果）。返回值的作用是接收出结果，使得它可以用于其他的操作！

我们可以按照方法的返回值和参数类型将方法分为下面这几种：

1.无参数无返回值的方法

```
public void f1() {  
    //.....  
}  
// 下面这个方法也没有返回值，虽然用到了 return  
public void f(int a) {  
    if (...) {  
        // 表示结束方法的执行，下方的输出语句不会执行  
        return;  
    }  
    System.out.println(a);  
}
```

2.有参数无返回值的方法

```
public void f2(Parameter 1, ..., Parameter n) {  
    //.....  
}
```

3.有返回值无参数的方法

```
public int f3() {  
    //.....  
    return x;  
}
```

4.有返回值有参数的方法

```
public int f4(int a, int b) {  
    return a * b;  
}
```

静态方法为什么不能调用非静态成员？

这个需要结合 JVM 的相关知识，主要原因如下：

1. 静态方法是属于类的，在类加载的时候就会分配内存，可以通过类名直接访问。而非静态成员属于实例对象，只有在对象实例化之后才存在，需要通过类的实例对象去访问。
2. 在类的非静态成员不存在的时候静态方法就已经存在了，此时调用在内存中还不存在的非静态成员，属于非法操作。

静态方法和实例方法有何不同？

1、调用方式

在外部调用静态方法时，可以使用 **类名.方法名** 的方式，也可以使用 **对象.方法名** 的方式，而实例方法只有后面这种方式。也就是说，**调用静态方法可以无需创建对象**。

不过，需要注意的是一般不建议使用 **对象.方法名** 的方式来调用静态方法。这种方式非常容易造成混淆，静态方法不属于类的某个对象而是属于这个类。

因此，一般建议使用 **类名.方法名** 的方式来调用静态方法。

```
public class Person {  
    public void method() {  
        //.....  
    }  
  
    public static void staicMethod(){  
        //.....  
    }  
  
    public static void main(String[] args) {  
        Person person = new Person();  
        // 调用实例方法  
        person.method();  
        // 调用静态方法  
        Person.staicMethod()  
    }  
}
```

2、访问类成员是否存在限制

静态方法在访问本类的成员时，只允许访问静态成员（即静态成员变量和静态方法），不允许访问实例成员（即实例成员变量和实例方法），而实例方法不存在这个限制。

重载和重写有什么区别？

重载就是同样的一个方法能够根据输入数据的不同，做出不同的处理

重写就是当子类继承自父类的相同方法，输入数据一样，但要做出有别于父类的响应时，你就要覆盖父类方法

重载

发生在同一个类中（或者父类和子类之间），方法名必须相同，参数类型不同、个数不同、顺序不同，方法返回值和访问修饰符可以不同。

《Java 核心技术》这本书是这样介绍重载的：

如果多个方法(比如 `StringBuilder` 的构造方法)有相同的名字、不同的参数，便产生了重载。

```
StringBuilder sb = new StringBuilder();
StringBuilder sb2 = new StringBuilder("HelloWorld");
```

编译器必须挑选出具体执行哪个方法，它通过用各个方法给出的参数类型与特定方法调用所使用的值类型进行匹配来挑选出相应的方法。如果编译器找不到匹配的参数，就会产生编译时错误，因为根本不存在匹配，或者没有一个比其他的更好(这个过程被称为重载解析(overloading resolution))。

Java 允许重载任何方法，而不只是构造器方法。

综上：重载就是同一个类中多个同名方法根据不同的传参来执行不同的逻辑处理。

重写

重写发生在运行期，是子类对父类的允许访问的方法的实现过程进行重新编写。

- 1. 方法名、参数列表必须相同，子类方法返回值类型应比父类方法返回值类型更小或相等，抛出的异常范围小于等于父类，访问修饰符范围大于等于父类。
- 2. 如果父类方法访问修饰符为 `private/final/static` 则子类就不能重写该方法，但是被 `static` 修饰的方法能够被再次声明。
- 3. 构造方法无法被重写

综上：重写就是子类对父类方法的重新改造，外部样子不能改变，内部逻辑可以改变。

区别点	重载方法	重写方法
发生范围	同一个类	子类
参数列表	必须修改	一定不能修改
返回类型	可修改	子类方法返回值类型应比父类方法返回值类型更小或相等
异常	可修改	子类方法声明抛出的异常类应比父类方法声明抛出的异常类更小或相等；
访问修饰符	可修改	一定不能做更严格的限制（可以降低限制）
发生阶段	编译期	运行期

方法的重写要遵循“两同两小一大”（以下内容摘录自《疯狂 Java 讲义》，[issue#892](#)）：

- “两同”即方法名相同、形参列表相同；
- “两小”指的是子类方法返回值类型应比父类方法返回值类型更小或相等，子类方法声明抛出的异常类应比父类方法声明抛出的异常类更小或相等；
- “一大”指的是子类方法的访问权限应比父类方法的访问权限更大或相等。

☆ 关于 **重写的返回值类型** 这里需要额外多说明一下，上面的表述不太清晰准确：如果方法的返回类型是 void 和基本数据类型，则返回值重写时不可修改。但是如果方法的返回值是引用类型，重写时是可以返回该引用类型的子类的。

```
public class Hero {
    public String name() {
        return "超级英雄";
    }
}

public class SuperMan extends Hero {
    @Override
    public String name() {
        return "超人";
    }
    public Hero hero() {
        return new Hero();
    }
}

public class SuperSuperMan extends SuperMan {
    public String name() {
        return "超级超级英雄";
    }

    @Override
    public SuperMan hero() {
        return new SuperMan();
    }
}
```

什么是可变长参数？

从 Java5 开始，Java 支持定义可变长参数，所谓可变长参数就是允许在调用方法时传入不定长度的参数。就如下面的这个 `printVariable` 方法就可以接受 0 个或者多个参数。

```
public static void method1(String... args) {
    //.....
}
```

另外，可变参数只能作为函数的最后一个参数，但其前面可以有也可以没有任何其他参数。

```
public static void method2(String arg1, String... args) {  
    //.....  
}
```

遇到方法重载的情况怎么办呢？会优先匹配固定参数还是可变参数的方法呢？

答案是会优先匹配固定参数的方法，因为固定参数的方法匹配度更高。

我们通过下面这个例子来证明一下。

```
/**  
 * 微信搜 JavaGuide 回复"面试突击"即可免费领取个人原创的 Java 面试手册  
 *  
 * @author Guide哥  
 * @date 2021/12/13 16:52  
 **/  
public class VariableLengthArgument {  
  
    public static void printVariable(String... args) {  
        for (String s : args) {  
            System.out.println(s);  
        }  
    }  
  
    public static void printVariable(String arg1, String arg2) {  
        System.out.println(arg1 + arg2);  
    }  
  
    public static void main(String[] args) {  
        printVariable("a", "b");  
        printVariable("a", "b", "c", "d");  
    }  
}
```

输出：

```
ab  
a  
b  
c  
d
```

另外，Java 的可变参数编译后实际会被转换成一个数组，我们看编译后生成的 `class` 文件就可以看出来了。

```
public class VariableLengthArgument {  
  
    public static void printVariable(String... args) {  
        String[] var1 = args;
```

```
        int var2 = args.length;

        for(int var3 = 0; var3 < var2; ++var3) {
            String s = var1[var3];
            System.out.println(s);
        }

    }
    // .....
}
```

基本数据类型

Java 中的几种基本数据类型了解么？

Java 中有 8 种基本数据类型，分别为：

- 6 种数字类型：
 - 4 种整数型：`byte`、`short`、`int`、`long`
 - 2 种浮点型：`float`、`double`
- 1 种字符类型：`char`
- 1 种布尔型：`boolean`。

这 8 种基本数据类型的默认值以及所占空间的大小如下：

基本类型	位数	字节	默认值	取值范围
<code>byte</code>	8	1	0	-128 ~ 127
<code>short</code>	16	2	0	-32768 ~ 32767
<code>int</code>	32	4	0	-2147483648 ~ 2147483647
<code>long</code>	64	8	0L	-9223372036854775808 ~ 9223372036854775807
<code>char</code>	16	2	'u0000'	0 ~ 65535
<code>float</code>	32	4	0f	1.4E-45 ~ 3.4028235E38
<code>double</code>	64	8	0d	4.9E-324 ~ 1.7976931348623157E308
<code>boolean</code>	1		false	true、false

对于 `boolean`，官方文档未明确定义，它依赖于 JVM 厂商的具体实现。逻辑上理解是占用 1 位，但是实际中会考虑计算机高效存储因素。

另外，Java 的每种基本类型所占存储空间的大小不会像其他大多数语言那样随机器硬件架构的变化而变化。这种所占存储空间大小的不变性是 Java 程序比用其他大多数语言编写的程序更具可移植性的原因之一（《Java 编程思想》2.2 节有提到）。

注意：

1. Java 里使用 `long` 类型的数据一定要在数值后面加上 `L`，否则将作为整型解析。
2. `char a = 'h'`:单引号，`String a = "hello"`:双引号。

这八种基本类型都有对应的包装类分别为：Byte、Short、Integer、Long、Float、Double、Character、Boolean。

基本类型和包装类型的区别？

- 成员变量包装类型不赋值就是 `null`，而基本类型有默认值且不是 `null`。
- 包装类型可用于泛型，而基本类型不可以。
- 基本数据类型的局部变量存放在 Java 虚拟机栈中的局部变量表中，基本数据类型的成员变量（未被 `static` 修饰）存放在 Java 虚拟机的堆中。包装类型属于对象类型，我们知道几乎所有对象实例都存在于堆中。
- 相比于对象类型，基本数据类型占用的空间非常小。

为什么说几乎是所有对象实例呢？ 这是因为 HotSpot 虚拟机引入了 JIT 优化之后，会对对象进行逃逸分析，如果发现某一个对象并没有逃逸到方法外部，那么就可能通过标量替换来实现栈上分配，而避免堆上分配内存

⚠ 注意：**基本数据类型存放在栈中是一个常见的误区！** 基本数据类型的成员变量如果没有被 `static` 修饰的话（不建议这么使用，应该要使用基本数据类型对应的包装类型），就存放在堆中。

```
class BasicTypeVar{
    private int x;
}
```

包装类型的缓存机制了解么？

Java 基本数据类型的包装类型的大部分都用到了缓存机制来提升性能。

Byte、Short、Integer、Long 这 4 种包装类默认创建了数值 `[-128, 127]` 的相应类型的缓存数据，Character 创建了数值在 `[0,127]` 范围的缓存数据，Boolean 直接返回 `True` or `False`。

Integer 缓存源码：

```
public static Integer valueOf(int i) {
    if (i >= IntegerCache.low && i <= IntegerCache.high)
        return IntegerCache.cache[i + (-IntegerCache.low)];
    return new Integer(i);
}
private static class IntegerCache {
    static final int low = -128;
    static final int high;
    static {
        // high value may be configured by property
        int h = 127;
    }
}
```

Character 缓存源码：

```
public static Character valueOf(char c) {
    if (c <= 127) { // must cache
```

```
        return CharacterCache.cache[(int)c];
    }
    return new Character(c);
}

private static class CharacterCache {
    private CharacterCache(){}
    static final Character cache[] = new Character[127 + 1];
    static {
        for (int i = 0; i < cache.length; i++)
            cache[i] = new Character((char)i);
    }
}
```

Boolean 缓存源码:

```
public static Boolean valueOf(boolean b) {
    return (b ? TRUE : FALSE);
}
```

如果超出对应范围仍然会去创建新的对象，缓存的范围区间的大小只是在性能和资源之间的权衡。

两种浮点数类型的包装类 `Float, Double` 并没有实现缓存机制。

```
Integer i1 = 33;
Integer i2 = 33;
System.out.println(i1 == i2); // 输出 true

Float i11 = 333f;
Float i22 = 333f;
System.out.println(i11 == i22); // 输出 false

Double i3 = 1.2;
Double i4 = 1.2;
System.out.println(i3 == i4); // 输出 false
```

下面我们来看一下问题。下面的代码的输出结果是 `true` 还是 `false` 呢？

```
Integer i1 = 40;
Integer i2 = new Integer(40);
System.out.println(i1==i2);
```

`Integer i1=40` 这一行代码会发生装箱，也就是说这行代码等价于 `Integer i1=Integer.valueOf(40)`。因此，`i1` 直接使用的是缓存中的对象。而 `Integer i2 = new Integer(40)` 会直接创建新的对象。

因此，答案是 `false`。你答对了吗？

记住：**所有整型包装类对象之间值的比较，全部使用 equals 方法比较。**

7. **【强制】**所有整型包装类对象之间值的比较，全部使用 equals 方法比较。

说明：对于 Integer var = ? 在 -128 至 127 之间的赋值，Integer 对象是在 IntegerCache.cache 产生，会复用已有对象，这个区间内的 Integer 值可以直接使用 == 进行判断，但是这个区间之外的所有数据，都会在堆上产生，并不会复用已有对象，这是一个大坑，推荐使用 equals 方法进行判断。

自动装箱与拆箱了解吗？原理是什么？

什么是自动拆装箱？

- **装箱：**将基本类型用它们对应的引用类型包装起来；
- **拆箱：**将包装类型转换为基本数据类型；

举例：

```
Integer i = 10;    //装箱
int n = i;         //拆箱
```

上面这两行代码对应的字节码为：

```
L1
    LINENUMBER 8 L1
    ALOAD 0
    BIPUSH 10
    INVOKESTATIC java/lang/Integer.valueOf (I)Ljava/lang/Integer;
    PUTFIELD AutoBoxTest.i :Ljava/lang/Integer;

L2
    LINENUMBER 9 L2
    ALOAD 0
    ALOAD 0
    GETFIELD AutoBoxTest.i :Ljava/lang/Integer;
    INVOKEVIRTUAL java/lang/Integer.intValue ()I
    PUTFIELD AutoBoxTest.n : I
    RETURN
```


从字节码中，我们发现装箱其实就是调用了包装类的 `valueOf()` 方法，拆箱其实就是调用了 `xxxValue()` 方法。

因此，

- `Integer i = 10` 等价于 `Integer i = Integer.valueOf(10)`
- `int n = i` 等价于 `int n = i.intValue();`

注意：如果频繁拆装箱的话，也会严重影响系统的性能。我们应该尽量避免不必要的拆装箱操作。

```
private static long sum() {  
    // 应该使用 long 而不是 Long  
    Long sum = 0L;  
    for (long i = 0; i <= Integer.MAX_VALUE; i++)  
        sum += i;  
    return sum;  
}
```

为什么浮点数运算的时候会有精度丢失的风险？

浮点数运算精度丢失代码演示：

```
float a = 2.0f - 1.9f;  
float b = 1.8f - 1.7f;  
System.out.println(a); // 0.100000024  
System.out.println(b); // 0.099999905  
System.out.println(a == b); // false
```

为什么会出现这个问题呢？

这个和计算机保存浮点数的机制有很大关系。我们知道计算机是二进制的，而且计算机在表示一个数字时，宽度是有限的，无限循环的小数存储在计算机时，只能被截断，所以就会导致小数精度发生损失的情况。这也就是解释了为什么浮点数没有办法用二进制精确表示。

就比如说十进制下的 0.2 就没办法精确转换成二进制小数：

```
// 0.2 转换为二进制数的过程为，不断乘以 2，直到不存在小数为止，  
// 在这个计算过程中，得到的整数部分从上到下排列就是二进制的结果。  
0.2 * 2 = 0.4 -> 0  
0.4 * 2 = 0.8 -> 0  
0.8 * 2 = 1.6 -> 1  
0.6 * 2 = 1.2 -> 1  
0.2 * 2 = 0.4 -> 0 (发生循环)  
...
```

关于浮点数的更多内容，建议看一下[计算机系统基础（四）浮点数](#)这篇文章。

如何解决浮点数运算的精度丢失问题？

`BigDecimal` 可以实现对浮点数的运算，不会造成精度丢失。通常情况下，大部分需要浮点数精确运算结果的业务场景（比如涉及到钱的场景）都是通过 `BigDecimal` 来做的。

```
BigDecimal a = new BigDecimal("1.0");
BigDecimal b = new BigDecimal("0.9");
BigDecimal c = new BigDecimal("0.8");

BigDecimal x = a.subtract(b);
BigDecimal y = b.subtract(c);

System.out.println(x); /* 0.1 */
System.out.println(y); /* 0.1 */
System.out.println(Objects.equals(x, y)); /* true */
```

关于 `BigDecimal` 的详细介绍，可以看看我写的这篇文章：[BigDecimal 详解](#)。

超过 long 整型的数据应该如何表示？

基本数值类型都有一个表达范围，如果超过这个范围就会有数值溢出的风险。

在 Java 中，64 位 long 整型是最大的整数类型。

```
long l = Long.MAX_VALUE;
System.out.println(l + 1); // -9223372036854775808
System.out.println(l + 1 == Long.MIN_VALUE); // true
```

`BigInteger` 内部使用 `int[]` 数组来存储任意大小的整形数据。

相对于常规整数类型的运算来说，`BigInteger` 运算的效率会相对较低。

参考

- What is the difference between JDK and JRE?: <https://stackoverflow.com/questions/1906445/what-is-the-difference-between-jdk-and-jre>
- Oracle vs OpenJDK: <https://www.educba.com/oracle-vs-openjdk/>
- Differences between Oracle JDK and OpenJDK:
<https://stackoverflow.com/questions/22358071/differences-between-oracle-jdk-and-openjdk>
- 彻底弄懂Java的移位操作符: <https://juejin.cn/post/6844904025880526861>